High-Level Synthesis of Multiple Dependent CUDA Kernels on FPGA

Swathi T. Gurumani[1], Hisham Cholakkal[1], Yun Liang[2], Kyle Rupnow[3], Deming Chen[4]

[1]Advanced Digital Sciences Center, Singapore
[2]Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, China
[3]Nanyang Technological University, Singapore
[4]University of Illinois at Urbana-Champaign, USA
[1]{swathi.g, hisham.c}@adsc.com.sg, [2]ericlyun@pku.edu.cn, [3]k.rupnow@ntu.edu.sg, [4]dchen@illinois.edu

Abstract— High-level synthesis (HLS) tools provide automatic generation of hardware at the register transfer level (RTL) from algorithm descriptions written in high-level languages, enabling faster creation of custom accelerators for FPGA architectures. Existing HLS tools support a wide variety of input languages, and assist users in design space exploration through automation and feedback on designs' performance bottlenecks. This design space exploration applies techniques such as pipelining, partitioning and resource sharing in order to improve performance, and resource utilization. However, although automated exploration can find some inherent parallelism, data-parallel input source code is still superior for exposing a greater variety of parallelism.

In prior work, we demonstrated automated design space exploration of GPU multi-threaded (CUDA) language source code for efficient RTL generation. In this paper, we examine the challenges in extending this automated design space exploration to multiple dependent CUDA kernels, demonstrate a step-by-step procedure for efficiently performing multi-kernel synthesis, and demonstrate the potential of this approach through a case study of a stereo matching algorithm. This study demonstrates that HLS of multiple dependent CUDA kernels can maintain performance parity with the GPU implementation, while consuming over 16X less energy than the GPU. Based on our manual procedure, we identify the key challenges in fully automating the synthesis of multi-kernel CUDA programs.

I. Introduction

FPGAs have long been used as an efficient implementation platform for application compute acceleration. FPGAs' reconfigurability provides implementation flexibility; designers for FPGA platforms optimize computation bit-width, computation block critical paths, and communication structures to create specialized, efficient hardware implementations. FPGA-based accelerators have been created for a variety of algorithms including compression, networking, cryptography, and video processing among many others. However, the availability of such hardware techniques comes at a cost: register transfer level (RTL) design time is much slower than algorithm implementations for CPU or GPU-based platforms. To resolve this problem, there have been decades of research in high-level synthesis (HLS): the process of automatically mapping high-level language code to RTL. Typically, HLS tools require restructuring of the original sequential software algorithm in C/C++ or reimplementation of the algorithm in a HLS-specific language in order to expose parallelism and facilitate the HLS process. Many such recent research efforts have been productive — with a large variety of both academic [1–8] and industrial [9–18] tools gaining increased acceptance. Collectively, these tools accept a wide range of different programming language inputs, such as functional programming languages [6, 7], Domain Specific Languages (DSLs) [17], extensions to C/C++ [1, 2, 11, 13, 18], GPGPU languages [3, 4, 8, 15], and graphical input languages [10].

In the same time-period as the rise of HLS tools, we have also experienced the emergence of general-purpose graphics processor (GPGPU) computation as a common compute-acceleration platform. NVIDIA released the data-parallel compute unified device architecture (CUDA) programming model [19] based on minimal extensions to C/C++ constructs to harness the parallel computational capabilities of GPUs. The wide adoption of the CUDA programming model has led to a large and growing set of compute-accelerators programmed and verified in a C-like language that is now widely used. CUDA's data parallel model exposes parallelism for easy application analysis. Thus, several academic projects for compute acceleration are based on CUDA: MCUDA maps CUDA code to multi-core processors and heterogeneous architectures [20] and FCUDA [3, 4] performs source-to-source translation of CUDA to AutoESL C code for HLS [18,21–23], allowing FPGA implementations that meet or exceed the GPU performance with significantly reduced power consumption. The data-parallel CUDA code allows both easier analysis of application parallelism and exploration of parallelism granularity options.

As a key feature of HLS, all tools offer some features to facilitate design space exploration: some choose programming language features or compiler directives to make design exploration simple for users, and others choose to automate (or partially automate) the selection of design parameters. In FCUDA-2 [4], design space exploration for a single CUDA kernel is automated, with generation of many design choices and selection of a design from the pareto-optimal set of designs based on an area-performance cost function. Prior tools also support automated or semi-automated design space exploration, and these tools can expand the optimization from a single C/C++ function to multiple functions simply by enclosing multiple functions in a wrapper function that describes communication between them. Similarly, FCUDA could also sup-

978-1-4673-3030-5/13/$31.00 ©2013 IEEE

port multi-kernel optimization through this path. It is possible to create a single enclosing wrapper kernel that uses the maximum thread dimension of all kernels and calls sub-kernels in succession. However, we argue that this implementation choice is neither attractive nor efficient for multiple reasons:

1. A single CUDA kernel must fully-buffer all sub-kernel communication on-chip — multiple CUDA kernels reduce buffering to the granularity of communication, and allows data streaming between sub-kernels for improved efficiency
2. A single CUDA kernel must use the same thread organization for sub-kernels whereas multiple CUDA kernels may each use different thread organization based on algorithm need
3. A single CUDA kernel forces all sub-kernels to be CUDA device-only functions — multiple CUDA kernels can be a mixture of device and host CUDA functions.

Therefore, in this work, we map multiple communicating kernels onto an FPGA architecture, allowing fine-grained communication and data streaming, different thread organizations, and full use of natural CUDA language features such as both *host* and *device* functions are fully synthesized. We first develop a process for using FCUDA-2 individually on multiple kernels and then manually connecting and optimizing the FCUDA-generated kernels through buffer insertion and resizing, pipelining and joint-design space exploration of the multiple kernels. Then, we use stereo matching, an active area of computer vision research, as a case study for evaluation of this process. Finally, we discuss the challenges in automating this manual process to automatically synthesize and perform design space exploration for multiple communicating CUDA kernels. This work contributes to the study of HLS with:

1. A demonstration of a manual step-by-step procedure to synthesize communicating CUDA kernels to RTL
2. An identification of key challenges to automate synthesis of multiple CUDA kernels to RTL
3. A case study of multiple CUDA kernel synthesis of a stereo matching algorithm

The rest of this paper is organized as follows: Section II presents an overview of FCUDA and our manual process for multiple kernel synthesis. Section III demonstrates the case study of applying our technique to a multi-kernel stereo matching source code, and section IV discusses the challenges in automating our manual multi-kernel synthesis process.

II. Multiple Dependent CUDA Kernel Synthesis

Our platform is based on FCUDA [3, 4], a source-to-source compilation framework that translates single-program multiple-data (SPMD) CUDA kernels to restructured C code for AutoPilot [18][1]. Then, AutoPilot is used to synthesize the C-code to synthesizable RTL, and the RTL is synthesized to an FPGA using Xilinx Vivado [22]. AutoPilot also generates a cycle-accurate SystemC simulation model that we use for functional verification.

FCUDA's source-to-source transformations convert data-parallel CUDA thread blocks to thread loops that can be mapped to FPGA hardware as custom compute engines (cores), and parallelism is explored by instantiating multiple cores. The FCUDA code transformations are guided by preprocessor directives to indicate how to decompose the FCUDA kernel into multiple levels of granularity that can be implemented efficiently in both area and latency [4]. The FCUDA framework can automatically generate and insert AutoPilot pragmas corresponding to automated design space exploration [4]. However, in the prior work, these automatic insertions assume that the kernel has all of the FPGA resources available to itself, and thus the selected solutions would not be based on joint optimization of multiple CUDA kernels. Thus, in this work, we allow FCUDA to insert the pragmas, but force the initial solution to be minimal in area; we will develop the analytical model for joint-design space exploration in the next sub-section.

A. Manual Process for Multiple Dependent Kernels

With multiple dependent CUDA kernels, FCUDA could directly inline kernels, but this doesn't allow sub-kernels to have different thread organizations, a feature that is natural and necessary for efficient implementation of communicating CUDA kernels. Therefore, instead of a single synthesis process for all of the kernels, we must individually synthesize the kernels, generate inter-kernel communication buffers and jointly optimize the kernels.

A.1 Individual Kernel Synthesis

The first step of the multiple dependent kernel synthesis is to individually synthesize each kernel using FCUDA and collect the resource consumption and latency metrics. Rather than the default FCUDA which attempts to maximize FPGA area usage in order to improve latency, we optimize to improve latency while minimizing single-core area in order to leave opportunity for joint-optimization of kernels later. For the minimal solution, which corresponds to one processing core, we measure the resource use and latency.

A.2 Communication Buffer Computation

With individually synthesized kernels, we will need to generate a control flow graph (CFG) for the kernels. This CFG is a simple graph of the kernels' dependence relations where we perform ASAP scheduling to determine the execution critical path and label the CUDA kernels that can execute in parallel. We need to insert buffers between each pair of communicating kernels. Ideally, we could buffer all data for the kernel and no partitioning would be necessary. However, this is not realistic in the general case; therefore we determine the size of the communication buffers after analyzing the data access patterns of the kernels. The analysis determines the minimal data processing quanta that retains the processing semantics to be passed between each pair of communicating kernels. In the analytical design space exploration, the algorithm will automatically

[1] AutoESL was acquired by Xilinx; AutoPilot is now part of Vivado

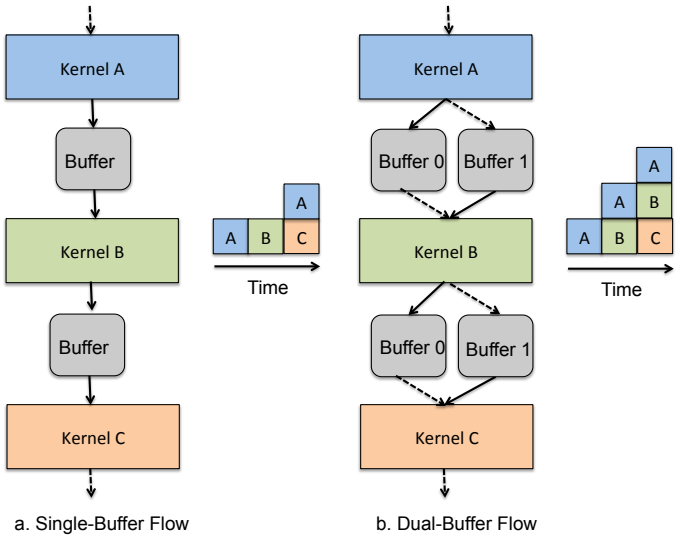

Fig. 1. Dual-Buffering Scheme to Enable Pipelining

explore larger buffer sizes when feasible. Therefore, the case of fully on-chip buffered data will also be discovered if that is feasible. We also determine the growth rate of the communication buffers as the data processing quanta is scaled. Note that for some communication patterns, there may be overlapping between communication windows. Thus, we model the communication buffer size using Eqn. (1). We map the communication buffers to BRAMs and this information is used in the analytical model.

$$BUF_{size} = comm\_size * nQuanta_i + overlap \quad (1)$$

We can implement the communication buffers in multiple ways: first, each set of kernels may have a single buffer between them. However, to ensure that data is not overwritten incorrectly, only one of the communicating kernels can use the buffer at a time (either read or write). This reduces pipelining throughput by a factor of 2, but can be effective if memory is a critical resource. The second implementation method uses double buffering between communicating kernels so that there is always a buffer available for reading and writing, which allows full pipelining throughput at the cost of an additional set of buffers. Our analytical model (next sub-section) models both buffering implementations and selects between them depending on resource limitations and achievable performance. A comparison of the buffering schemes is shown in Figure 1.

A.3 Analytical Design Space Exploration Model

In order to analytically model the performance of a sequence of dependent CUDA kernels, we use several input data values: the resource consumption and latency of each individual synthesized kernel, control flow graph of the dependence relations of the CUDA kernels, the resource consumption of communication buffering schemes, and total available resources of the target FPGA.

With the scheduled CFG, we can now build a simple analytical model for the performance of a jointly-optimized set of dependent CUDA kernels. Let us define that we have N kernels k_0 to k_{N-1}. Each kernel k_i has a latency lat_i, a predence level in the CFG lev_i, a minimal communication quanta $nQuanta_i$ (in units of CUDA thread blocks), and a 4-tuple of its resource use $(BRAM_i, FF_i, LUT_i, DSP_i)$. The number of levels in the CFG determines the number of pipeline stages in the design (and the analytical model). The worst-case pipeline stage latency is simply the maximum latency of any kernel in the graph as in Eqn. (2).

$$lat_{crit} = MAX(lat_i) \quad (2)$$

For each kernel, based on the number of cores available for computation, we can compute the number of execution *phases* within each epoch as in Eqn. (3). Then, the latency of the kernel lat_i is the simple product of the latency of a single thread block and the number of phases. Note that our analytical exploration forces the number of allocated cores to produce integral values of $phase_i$; alternatively, you could formulate Eqn. (3) as the ceiling of the given fraction.

$$phase_i = \frac{nQuanta_i}{cores_i} \quad (3)$$

$$lat_i = lat_{TB} * phase_i \quad (4)$$

Based on the communication quanta size, we can compute the number of epochs as

$$N_{epoch} = tot_{TB}/nQuanta \quad (5)$$

The value of nQuanta is computed such that N_{epoch} is an integral value. Additionally, because nQuanta is computed as a quanta that selects a coherent data processing unit for all kernels, N_{epoch} can be computed using the total thread blocks and quanta of any kernel in the design (for simplicity, we use the first kernel). Given these values, we now have the latency of each kernel as a function of the number of FPGA resources allocated to it, the number of stages in the design's pipeline, and the number of quanta that must be sent through the pipeline. Therefore, the total design latency for the double buffered communication style is simply computed by Eqn. (6). For the single buffered communication, the pipeline depth remains the same, but lower throughput effectively doubles the number of epochs.

$$Latency = (N_{epoch} + pipe\_depth - 1) * lat_{crit} \quad (6)$$

As we evaluate different communication quanta and cores per kernel, the critical path and N_{epoch} will change, leading to different total latencies in the design space. Note that in the general case, we can also consider splitting each kernel into multiple pipeline stages, which would correspondingly affect pipe_depth and lat_{crit}. In the FCUDA framework, we could extend to splitting kernels by allowing each of the *COMPUTE* blocks to be a pipeline stage, or combine kernels with identical thread structure to merge pipeline stages. However, we leave this extension to future work.

With the above computation, we model the performance of any particular partitioning of a total computation into epochs and allocation of computation resources to the kernels. Each kernel's resource use 4-tuple is used to compute the maximum (resource limited) implemented cores using Eqn. (7). In practice, this upper limit is much larger than the number of thread

blocks in the quanta so we compute the final maximum implemented cores using Eqn. (8).

$$Res\_Lim_i = MIN(\frac{BRAM}{BRAM_i}, \frac{FF}{FF_i}, \frac{LUT}{LUT_i}, \frac{DSP}{DSP_i}) \quad (7)$$

$$Lim_i = MIN(Res\_Lim_i, nQuanta_i) \quad (8)$$

These limitations of kernel core implementation set some bounds on the analytical design space search. We compute the total resource simply as the sum of kernels' resource use and communication buffers' resource use. Together, we can now use the resource use and latency to perform an analytical search of the design space as follows:

Algorithm 1: Analytical Design Exploration Model

```
let D be the minimal nQuanta for the design (buffers) ;
let C be the set of per-kernel core allocations ;
let A be the area of C plus communication buffers D ;
best_lat = INF ;
best_soln = (C, D) ;
while A < FPGA_resources do
    while A < FPGA_resources do
        find kernel k_i that has lat_i == lat_crit ;
        if C_i < nQuanta_i then
            phase = nQuanta_i / C_i ;
            C_i = nQuanta_i / (phase--) ;
            compute A and lat_total values ;
            if A < FPGA_resources && lat_total < best_lat
            then
                best_lat = lat_total ;
                best_soln = (C, D) ;
            end
        end
        else
            no core increase reduces critical path ;
            break;
        end
    end
    increment D to next value with coherent, integral D_i values ;
    set C to the minimal core allocation at nQuanta D ;
    compute A value ;
end
```

After using this exploration algorithm, the final solution ($best\_soln$) will contain a set of suggested core allocations and nQuanta values (that determine the total number of epochs).

A.4 Implementation and Verification

Using the analytically computed core allocation values, we implement the suggested core allocations and communication buffer parameters and synthesize the resulting AutoPilot-C code to RTL. We update the AutoPilot-C pragmas to correspond to the suggested parallelism, insert the communication buffers and add proper pragmas to ensure that the communication buffers support pipelined computation at the kernel granularity. The RTL is then synthesized to the target FPGA and we verify that the design fits and operates correctly.

III. Case-Study: Stereo Matching

We have now presented our manual method and analytical model for synthesizing multiple dependent CUDA kernels using the FCUDA and AutoPilot tools. Now, we present a case study of applying this method to a real application with multiple dependent CUDA kernels. Stereo matching [24, 25] algorithms estimate depth by comparing two or more time-synchronized but spatially separated images and measuring the disparity between corresponding points in the images. The algorithm selected for this case study uses Bilateral Filtering with an Adaptive Support (BFAS) function for a local optimization stereo matching method. In prior work, we demonstrated that there is a performance gap between synthesizing stereo matching algorithms written in C using AutoPilot and manual FPGA implementations of stereo matching [26].

In order to meet real-time requirements using a Point-Grey stereo-camera system [27], we implemented the BFAS algorithm in CUDA on an NVIDIA GTX480. The CUDA implementation used as input to FCUDA in this paper consists of 8 sub-kernels (some sub-kernels are called twice, once each for the left- and right-images), and can meet real-time performance for 720x1280 (720p) images arriving at 15 frames per second (the maximum frame rate of our stereo-camera). The eight sub-kernels are census transform, RGB to Lab colorspace conversion, left grid building, right grid-building, matching, cross correction, pre-filtering, and median filtering; for more information on algorithmic details see [24]. We will now apply our manual process for synthesizing these dependent kernels.

A. Step 1: Individual Kernel Synthesis

The eight individual kernels are extracted from the CUDA implementation and each kernel is independently synthesized using the FCUDA tool. Note that the left grid and right grid processing kernels are logically similar, but their computations are sufficiently different that they require different hardware implementations; thus, the grid building kernel is the only kernel that is individually specialized for left- or right-image processing, although (as we see explicitly in the next subsection) many of the kernels are called twice. The original CUDA implementation includes floating point computation and dynamic memory allocation for sizing the image buffer; we convert to fixed point computations and a static image size in order to support HLS.

For each of these kernels we perform the FCUDA source-to-source translation, AutoPilot C-to-RTL synthesis, and Vivado RTL synthesis to a Virtex-7 XC7VX1140T to gather area and performance estimates. We note that the AutoPilot synthesis estimates for latency are only accurate in the absence of dataflow pipelining, and with statically determined loop trip counts. However, the FCUDA translation of CUDA to C often interferes with AutoPilot's ability to estimate latency in clock cycles. We verified that the AutoPilot SystemC model simulation is accurate with respect to the latency in clock cycles of the synthesized RTL, therefore we use that SystemC simulation to determine each kernel's latency. Table I shows the synthesis results and latency in clock cycles for each of the sub-kernels.

TABLE I
KERNEL SYNTHESIS DATA

Kernel Name	DSP	FF	LUT	BRAM	Clock Cycles)
Census	3	450	1054	0	2917
RGB to LAB	41	935	1150	18	5221
Left Grid	3	1498	2333	2	63011
Right Grid	9	1588	2518	2	61011
Matching	37	2420	4240	11	662371
Cross Corr.	3	397	537	0	1765
Prefiltering	4	584	852	2	40914
Median Filter	3	364	432	0	1759

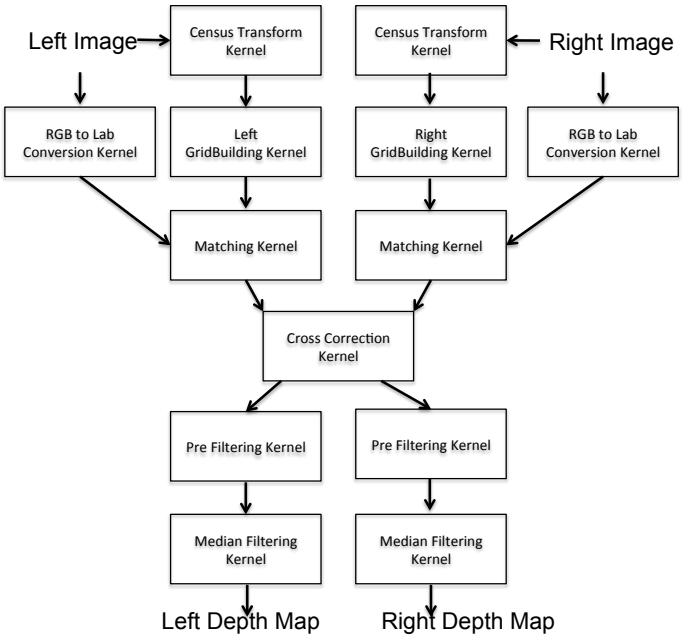

Fig. 2. CUDA Kernels Representing the Transforms of Stereo Matching Algorithm

B. Step 2: Communication Buffers

As the first step of the determining necessary communication buffers, we generate a CFG of the kernels in the stereo matching application. Figure 2 depicts the set of transforms and the dependence relations between them. This CFG will guide our pipeline implementation and performance model in the next section when finding candidate solutions with the analytical model. The buffer size and computation quanta are selected such that one epoch of data provided to the first kernels in the CFG corresponds to an integer number of thread blocks for every kernel in the control flow graph. For the stereo matching application, we determine that due to the restrictions on computation correctness, thread organization, and buffering quanta, the minimum sub-image size is a 6x96 pixel sub-image.

Table II shows the buffer size in BRAMS for the minimal buffer size for each of the communication paths. Note that, according to the CFG some of the paths are identical and duplicated (e.g. 2 buffers between the Cross Correction kernel and the 2 copies of the pre-filtering kernels). In addition, as discussed above, there are both single- and double-buffered schemes that allow different amounts of parallelism. Double buffering schemes consume twice as many BRAMs in exchange for an increase in pipeline throughput.

TABLE II
COMMUNICATION BUFFER DATA

Communication Buffer	BRAM
Census ⇒ Left Grid	3
Census ⇒ Right Grid	3
Left Grid ⇒ Matching	8
Right Grid ⇒ Matching	8
RGB to LAB ⇒ Matching	3
Matching ⇒ Cross Correction	1
Cross Correction ⇒ Prefiltering	1
Prefiltering ⇒ Median filtering	1

C. Step 3: Analytical Design Exploration

Given the area and latency data for each kernel, the buffers between them and the CFG of dependencies, we can now use the analytical model to determine the solution that fills the FPGA while minimizing the total latency of the design. Intuitively, the analytical model is aware that the buffering between kernels consumes resources, but does not necessarily reduce the compute latency. Therefore, the model balances between solutions that fill the FPGA with cores from kernels (to minimize critical path) and adding more buffering to allow more parallel processing opportunity.

Figure 3 shows, for the stereo matching application, the area and latency of every candidate solution evaluated by the analytical model for the dual-buffer (DB) implementation. The best feasible solution in the design space is annotated. Every point in Figure 3 is feasible point with individual resource use of each resource type less than or equal to 100% of available resources. In this graph, area is computed as the sum of each resource type's percentage of resources used; therefore, area values are normalized to be between 0.0 and 4.0, with all infeasible combinations eliminated. This analytical design space exploration demonstrates several expected trends: reducing latency as resource use increases at any particular buffer size, and increasing latency and resource use with identical computation cores as buffer size grows. Thus, although a larger buffer size increase the number of core implementation options, it may also increase the latency such that increased cores cannot overcome the additional overheads of larger communication and data processing quanta. Figure 4 shows the design space for both the single buffer (SB) and dual-buffer (DB) implementations. Figure 5 shows an example of the code modifications necessary to introduce the ping-pong (dual buffering) scheme.

Based on the analytical modeling, we select a solution with a sub-image buffer of 12x96, double buffering, 36 parallel cores for the matching transform, 4 cores for grid building and 3 cores for pre-filtering transform in order to reduce the critical path. Based on this selected design, we set the AutoPilot HLS parameters, generate RTL and synthesize the design to gather final area and clock period. The resources usage for the design are DSP: 2833(85%); FF: 193983(14%); LUT: 335605(47%);

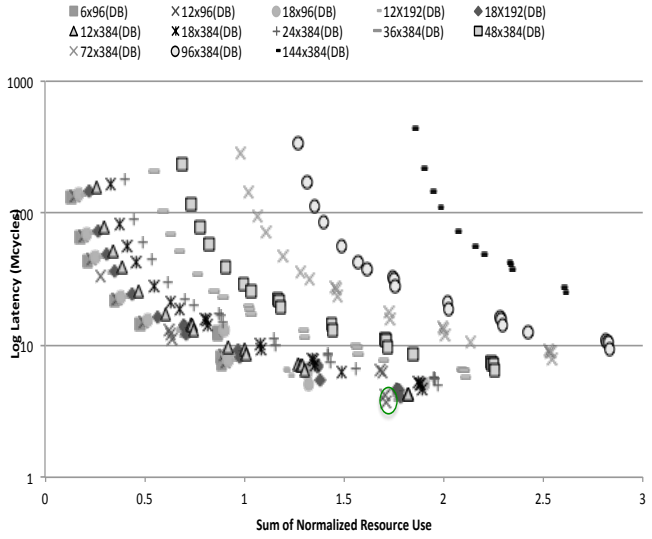

Fig. 3. Design Space Evaluated by the Analytical Model for Dual-Buffer Flow

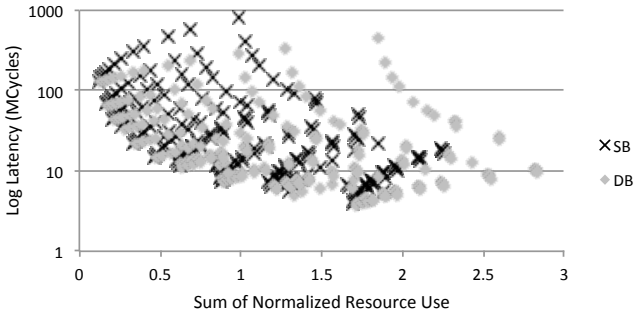

Fig. 4. Design Space Showing both Dual and Single Buffer Flow

BRAM: 992(25%). The design was synthesized to clock period of 7.5ns. Performance and energy comparison between the GPU and FPGA solution is shown in Figure 6. We used the average power consumption of GPU GTX480 as 223W and used the Xilinx Power Estimator tool to calculate FPGA power consumption as 13.4W for a savings of over 16X. Table III shows the latency and power metrics for GPU and FPGA.

TABLE III
PERFORMANCE AND POWER RESULTS

Architecture	Latency (ms)	Power (W)
GPU	29	223
FPGA	27.8	13.4

D. Observations

The analytical model does an effective job of pruning the design space through iteratively increasing resource allocation to kernels on the critical path, and increasing the communication

```
void stereo_top (unsigned char left_img_ping[I], unsigned char left_img_pong[I],
unsigned char right_img_ping[I], unsigned char right_img_pong[I],unsigned
short left_depth_ping[I], unsigned char left_depth_pong[I]]) {
#pragma AP dataflow
…
static int ping_pong = 0;
unsigned char census_left_ping[image_size]; //Intermediate buffer declarations
unsigned char census_left_pong[image_size]; //for ping-pong
unsigned char left_grid_disparity_ping[disp_image_size];
unsigned char left_grid_disparity_pong[disp_image_size];
…
if (ping_pong == 0) {                                          read
censusTransformKernel_left (left_img_ping, census_left_pong);
leftGridBuildKernel_left(census_left_ping,left_grid_disparity_pong,
left_grid_cost_pong);                                                  write
…
ping_pong = 1;}
else{                                                          read
censusTransformKernel_left (left_img_pong, census_left_ping);
leftGridBuildKernel_left(census_left_pong, left_grid_disparity_ping,
left_grid_cost_ping);                                                  write
…
ping_pong = 0;}
}
```

Fig. 5. Code Modifications to Introduce Dual-buffering Scheme

buffering when parallelism opportunity is limited by the communication quantum. The solution is able to meet performance parity with a GPU. This presents an interesting case: when synthesizing the original C code for the BFAS algorithm [26], we achieved a 6.9X speedup over the original source code. However, the GPU implementation sped up the software by over 50X in order to meet real-time constraints and our synthesis of this implementation is able to maintain performance parity. This demonstrates the benefits of a data-parallel input language for HLS: greater exposed parallelism gives the synthesis tool greater opportunity for optimization while still meeting both performance and power/energy constraints.

Multiple dependent CUDA kernel synthesis is thus an attractive implementation platform for HLS of application accelera-

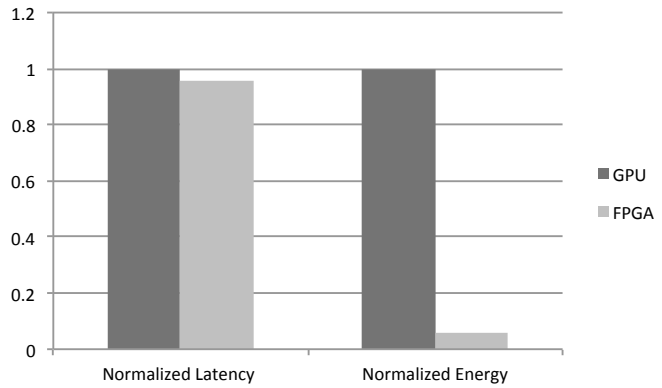

Fig. 6. GPU-FPGA Comparison

tors. The additional benefit of uniform programming interface with GPU platforms makes CUDA implementation even more attractive. However, although the FCUDA tool assisted in this achievement, it is important to note that this manual process still requires some effort in the mapping process and there will be challenges in fully automating this flow for multiple dependent CUDA kernels.

IV. Challenges: Multi-kernel CUDA Programs to Hardware

We have identified some challenges for fully automating our manual design flow for multiple dependent CUDA kernel synthesis. We divide the challenges into two main categories: 1. challenges in improving FCUDA individual kernel synthesis, and 2. challenges in multi-kernel analysis. Although the first category of challenges are primarily for single kernel synthesis, the fact that we are duplicating the initial solutions many times makes the efficiency of single kernel synthesis even more critical than in the original FCUDA implementation.

A. Single Kernel Synthesis

In realistic CUDA kernels, thread indexing is often a complex combination of the block and grid dimensions. Simple data-parallel indexing computation is supported efficiently by FCUDA, but more complex indexing schemes can both increase computation resources to implement index calculation and increase the difficulty of determining access orders, data dependencies, and loop iteration bounds. For FCUDA, improved analytical techniques to optimize index computations and/or transform access patterns to lookup tables could significantly improve the resource use per kernel and thus the design space exploration flexibility. Furthermore, improved optimization of index computations will also improve the opportunity to apply other optimizations such as array partitioning.

Similarly, many applications (such as the stereo matching) use constant multiplications and divisions for signal processing computations. When these constants are powers of two, FCUDA and AutoPilot correctly convert the operation to shifts with little resource consumption. However, other constants generate inefficient implementations that consume many DSP units — efficient alternative implementations can significantly reduce resource consumption (e.g. table-based constant dividers [28]).

Finally, FCUDA and AutoPilot do not currently support automatic translation from floating-point to fixed-point computation. Transformation to fixed-point computation can be particularly beneficial to meet area constraints and thus allowing more instantiated parallelism. However, it is difficult to automatically determine if such a transformation is feasible, functionally correct or desirable for the implementer.

B. Multiple Kernel Synthesis

The first challenge in multiple kernel synthesis relates to the selection of the initial single-core implementation for each of the sub-kernels. Because these single-core implementations will be duplicated many times, it is important to select an area efficient implementation. In this work, we do this manually, but in an automatic flow this would require a complex value function and iteration of the entire design flow to improve the single-core latency of critical kernels. Additionally, this would require integrated knowledge of critical resource use so that optimization decisions can effectively decide whether an optimization impacts implementation choices for the multi-kernel design.

The next challenge in multiple kernel synthesis is the automatic buffer generation and insertion. This process requires complex memory access pattern analysis to determine what data is required by groups of threads, the access order, overlap between adjacent regions, and ability to partition the computation into sequential independent computations. This needs to be done individually for each pair of communicating kernels and then collectively among all kernels to find a least-common-multiplier of buffer sizes that meets all of the access limitations, communication quantum limitations, and partitioning limitations. In the manual flow, we are able to use designer knowledge of the algorithm to assist in this analysis, but an automated flow will require significant improvements to statically guarantee all these properties.

Another significant challenge in multi-kernel synthesis relates to performance estimation within the synthesis process. In this work, we use full AutoESL synthesis followed by SystemC simulation of produced designs in order to gather latency information for individual kernel choices, and then we extrapolate those latencies based on buffer sizes and dividing the buffer into a sequence of operations. However, in a fully automated system, it is more desirable to analytically determine candidate performance without performing AutoESL synthesis and simulation for any candidate component. These performance modeling improvements will require improved analytical abilities for determining loop bounds and trip counts in situations in which loop parameters are statically determinable through simple analytical modeling.

Finally, a major challenge in multi-kernel synthesis relates to sub-kernel optimizations that could be performed in order to better match pipeline stage latency throughout the design. In this work, we pipeline each stage, but there is no implementation to combine two or more small pipeline stages into a single stage or divide a large pipeline stage into two or more stages. These optimizations increase complexity by requiring the synthesis tool to automatically select synchronization points and generate buffers between the new stages. In some cases, it needs to perform the data access pattern analysis to allow multiple stages to be combined with intervening buffers merged. However, with this ability, multi-kernel optimizations could be yet more efficient by automatically generating a well-balanced pipeline implementation.

V. Conclusion

We have demonstrated a manual process for mapping multiple dependent CUDA kernels to an FPGA using the FCUDA tool that acheives performance parity with GPUs while consuming over 16X less energy than the GPU. Our analytical model for simultaneous design space exploration of multiple kernels allows efficient implementation decisions that balance

the communication resources and computation resources in order to find design points that effectively use all FPGA resources to best minimize total computation latency. Through a case study of applying this technique on a multi-kernel implementation of a stereo matching application, we have demonstrated the potential for multi-kernel synthesis: starting with a data-parallel input language, we can achieve performance parity with GPUs with significantly reduced energy consumption, and this implementation also represents a significant speedup over HLS of a sequential C++ implementation of the same algorithm. Finally, through this case study, we have identified the key challenges in fully automating this multi-kernel process.

VI. ACKNOWLEDGEMENT

This work was supported by A*STAR under the HSSP grant.

REFERENCES

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Symposium on Field Programmable Gate Arrays (FPGA)*. New York, NY, USA: ACM, 2011, pp. 33–36.

[2] H. Zheng, Q. Liu, J. Li, D. Chen, and Z. Wang, "An open source high-level synthesis framework with cross-level optimizations." [Online]. Available: https://github.com/SysuEDA/Shang

[3] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. mei W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Symposium on Application Specific Processors (SASP)*, 2009, pp. 35–42.

[4] A. Papakonstantinou, Y. Liang, J. A. Stratton, K. Gururaj, D. Chen, W. mei W. Hwu, and J. Cong, "Multilevel Granularity Parallelism Synthesis on FPGAs," in *Field-Programmable Custom Computing Machines (FCCM)*, 2011, pp. 178–185.

[5] D. Greaves and S. Singh, "Kiwi: Synthesis of fpga circuits from parallel programs," in *Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2008, pp. 3–12.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *ACM SIGPLAN Notices*, vol. 34, no. 1. ACM, 1998, pp. 174–184.

[7] C. Baaij, M. Kooijman, J. Kuper, W. Boeijink, and M. Gerards, "Cλash: Structural descriptions of synchronous hardware using haskell," *Digital Systems Design (DSD)*, pp. 714–721, 2010.

[8] M. Lin, I. Lebedev, and J. Wawrzynek, "Openrcl: low-power high-performance computing with reconfigurable devices," in *Field Programmable Logic and Applications (FPL)*. IEEE, 2010, pp. 458–463.

[9] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Formal Methods and Models for Co-Design (MEMOCODE)*. IEEE, 2004, pp. 69–70.

[10] *LabVIEW FPGA IP Builder*, National Instruments, http://sine.ni.com/nips/cds/print/p/lang/en/nid/210573/.

[11] *ImpluseC*, Impulse Accelerated Technologies, www.impulseaccelerated.com.

[12] *Catapult C Synthesis*, CALYPTO, 2012, http://www.calypto.com/catapult_c_synthesis.php.

[13] *Cynthesizer*, Forte Design Systems, 2012, http://www.forteds.com/products/cynthesizer.asp.

[14] *SystemC*, Accelera Systems Initiative, www.accelera.org.

[15] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh, "From opencl to high-performance hardware on fpgas," in *Field Programmable Logic and Applications (FPL)*. IEEE, 2012, pp. 531–534.

[16] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a java-compatible and synthesizable language for heterogeneous architectures," in *Object oriented programming systems languages and applications (OOPSLA)*. New York, NY, USA: ACM, 2010, pp. 89–108.

[17] B. Bond, K. Hammil, L. Litchev, and S. Singh, "Fpga circuit synthesis of accelerator data-parallel programs," in *Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2010, pp. 167–170.

[18] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "Autopilot: A platform-based esl synthesis system," *High-Level Synthesis: From Algorithm to Digital Circuit*, pp. 99–112, 2008.

[19] *CUDA Parallel Computing Platform*, NVIDIA, 2012, http://www.nvidia.com/object/cuda_home_new.html.

[20] J. A. Stratton, S. S. Stone, and W. mei W. Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, Languages and Compilers for Parallel Computing," in *Languages and Compilers for Parallel Computing (LCPC)*, 2008, pp. 16–30.

[21] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xpilot: A platform-based behavioral synthesis system," *SRC TechCon*, vol. 5, 2005.

[22] *Vivado High-Level Synthesis*, Xilinx Inc, http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/hls/index.htm.

[23] *AutoESL Reference Manual*, Xilinx Inc, www.xilinx.com.

[24] D. M. Dongbo Min, Jiangbo Lu, "A revisit to cost aggregation in stereo matching: How far can we reduce its computational redundancy?" in *IEEE International Conference on Computer Vision(ICCV)*, 2011, pp. 1567–1574.

[25] D. B. Min and K. Sohn, "Cost Aggregation and Occlusion Handling With WLS in Stereo Matching," *IEEE Transactions on Image Processing*, vol. 17, pp. 1431–1442, 2008.

[26] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *Field-Programmable Technology (FPT)*. IEEE, 2011, pp. 1–8.

[27] *Point Grey Stereo Vision Cameras*, Point Grey Research, http://www.ptgrey.com/products/stereo.asp.

[28] F. De Dinechin and L. Didier, "Table-based division by small integer constants," *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 53–63, 2012.